Tema 4: Hash tabeller

Henrik Thulin `heth7132`
Mattin Lotfi `malo5163`

14 februari 2016

1 Hashtabeller

1.1 Hashing

Vill man hantera mängder av dataobjekt kan man behöva inkludera dessa i en gemensam datastruktur. Några exempel på datastrukturer är länkade listor, arrayer och binärträd. Till det kommer även hashtabeller vilket denna artikel avser att introducera. Gemensamt för dessa strukturer är att de inkluderar möjligheten till att kunna lägga till, ta bort och söka specifikt dataobjekt. Effektivitet för de processer som krävs för att kunna utföra dessa funktioner skiljer sig dock nämnvärt.

En sökning för en osorterad array är linjär. Det innebär att man kan behöva gå igenom hela datamängden för att kunna finna det man söker, om det nu överhuvudtaget existerar. Det

gäller även länkade listor. Sökningen blir tidskrävande och därför är binärträd och sorterad arrayer att föredra då man kan implementera sökmetoder som sker logaritmisk. Sökningen halverar den mängd som ingår vid varje miss tills objekt blivit funnet och sker därmed snabbare.

Att behöva iterera genom långa listor tar tid och det är här fördelen med hashtabeller ligger. Hashtabeller arbetar, till skillnad från ovan exempel, med nycklar och kräver inte att tabellen är sorterad för att snabbt kunna lokalisera eller sätta in objekt. För att detta ska kunna fungera behöver de objekt som hashtabellen förvarar kunna avge en hashkod. Hashkoden används för att identifiera objektet och fungerar som objektets personliga signatur. Denna kod räknas om för att anpassas till hashtabellens storlek och användas som nyckel för att bestämma var i tabellen objektet bör placeras. Det är inte alltid objektet hamnar på nyckelns position.

Eftersom längden (eller kapaciteten) på tabellen begränsar möjliga positioner finns det en risk att det uppstår en kollision. Detta resulterar i att man inte alltid lyckas genomföra en sökning eller en insert på konstant tid. Säg att vi har ett objekt vars hashkod är 123 och ett annat som är 23. Med en tabellstorlek på 100 så skulle t.ex. båda, då 123 modulo 100 och 23 modulo 100, kunna hamna på samma position. M.a.o. så kan uträkningen för flera olika hashkoder resultera i samma nyckel i hashtabellen.

1.2 Open hashing

För att hantera kollisioner kan positionen som beräkningen av hashkoden resulterar i istället referera till en understruktur bestående av en lista, t.ex. en länkad lista, där de element som kolliderat lagras. Denna metod går under benämningen separate chaining (kallas även open hashing) och i den underliggande listan finns de element vars hashkod motsvarar samma nyckel. Denna lista kan vara en länkad lista, binärträd eller liknande samling. Vilken som lämpar sig bäst avgörs av den typ av information man avser att hantera och en avvägning av resurstillgång och effektivitetsbehov. Generella fördelar med metoden är att den är relativt enkel att implementera och kan lagra ett stort antal kollisioner. En full hashtabell kräver därför inte en omdimensionering. Är objekten som lagras tillräckligt stora så blir inte den overhead som uppstår i listor, d.v.s. utrymmet som krävs av elements olika pekare, något större problem. Skulle däremot objekten vara små kan en betydande del av varje elements storlek påverkas negativt och kräver i proportion överdrivna minnesresurser. En annan aspekt är att eftersom nya strukturer behöver allokeras externt från hashtabellen kan det påverka accesstiden och minneshanteringen.

1.3 Closed hashing

Ett alternativt sätt som även går under namnet open adressing innebär att objekten lagras direkt i hashtabellen. Positionen bestäms av nyckeln och efter en viss beräkning beroende på vilken teknik man väljer att tillämpa. Eftersom kollisioner kan upp-

stå hamnar inte alltid objektet på den plats där nyckeln pekar. Man behöver längre inte lämna tabellen utan kan lokalt hantera objekten vilket kan ha fördelar när det kommer till hur cache bearbetas såväl som accesstid. Det kräver heller inte pekare som i fallet med listor.

1.3.1 Linear probing

Med en metod kallad linear probing så, vid en kollision, undersöks efterföljande eller ett annat bestämt avstånds utrymme istället. Skulle även den vara ockuperad provas nästa, och på så sätt fortsätter det tills ett ledigt utrymme uppkommer där ett nytt objekt kan placeras. Precis samma princip används vid en sökning och ett ledigt utrymme markerar slutet på sökningen. Ett problem med denna metod är primary clustering. Kluster uppstår när flera kollisioner inträffat och fält i sekvenser därför blivit ockuperade. Dessa kluster har en förmåga att växa och sammanlänkas med andra. Den linjär genomgång av tabellen vid en insert eller search kan därför kräva att en stor del av, eller hela, tabellen måste genomsökas, och är metoden felimplementerad kan detta pågå för evigt. Det finns även en risk att en insert inte kan genomföras p.g.a. att tabellen blivit full. Förutsätter vi att load factor, vilket kännetecknas av hur stor andel av tabellens platser är upptagna, inte är för hög så är inserts och searches generellt snabba men påverkas snabbt när faktorn ökar.

1.3.2 Quadratic probing

Quadratic probing är en liknande metod som faller under samma kategori och kan användas för att lösa problemet med primary clustering. Den fungerar på ett liknande sätt men placerar istället ut de objekt som kolliderat med ett exponentiellt, d.v.s. i kvadrat, ökande avstånd. Däremot så kan istället ett problem vid namn secondary clustering uppstå där vissa nycklar börja mappa mot samma element. Därför är det viktigt att tabellens storlek är ett primtal samt att den är mindre än halvfull för att kunna garantera en ledig plats för en ny insättning. Detta kräver därför att mycket av de allokerade cellerna står oanvänt för att vara en väl fungerande lösning.

1.3.3 Double hashing

Består av två hashfunktioner. Den första positionerar elementet i listan och den andra definierar hur eventuella hopp vid en kollision ska ske. En viktig egenskap hos den andra hashfunktionen är att den ej får evaluera till 0. Den bör heller se till att samtliga positioner i tabellen kan undersökas.

Denna teknik motverkar secondary clustering då risken minskar för att de element som beräknats till en viss position agerar lika efter en kollision. En fördel med denna typ av lösning är att load factor kan vara högre än t.ex. linear probing utan att prestandan för en insert påverkas i lika stor grad. Däremot kan uträkningen kräva en större ansträgning

Även här är det viktigt att tabellens storlek är ett primtal, detta för att säkerställa att maximalt antal positioner i följd kan genomsökas.

1.3.4 Rehashing

Skulle tabellerna bli fulla kan man utföra en omdimensionering eller en så kallad rehashing. Det innebär att man gör tabellen större genom att kopiera posterna från den tidigare till en ny som ofta är dubbelt så stor och ett primtal. I den nya tabellen placeras objekten på lämpliga positioner efter den metod man valt att använda.

Ett problem med de tekniker som nämnts berörande closed hashing är att man inte helt kan radera ett objekt i tabellen utan att påverka de steg som kan krävas för att lokalisera ett tomt utrymme eller finna det objekt man eftersöker när kedjan bryts. Istället krävs därför endast en lazy deletation där objektet markeras som oanvänt men finns kvar som en ”placeholder”. Dessa använda men oanvända positioner tas bort i en rehashing.

Rehashing processen kan exempelvis genomföras efter antal använda poster överskrids, d.v.s. load factor överstiger ett gränsvärde, eller vid ett tillfälle där man inte lyckats införa ett nytt objekt. För quadratic probing förutsätts det att mer än halva tabellen är ledig för inserts ska kunna fungera korrekt.

2 Frågor

2.1 Fråga 1

Varför uppstår kluster?

2.2 Fråga 2

Varför bör tabellen i vissa fall vara ett primtal?